METHOD AND APPARATUS FOR MANAGING SOFTWARE COMPONENT DOWNLOADS AND UPDATES

FIELD OF THE INVENTION

[01] This invention relates to methods and apparatus for software distribution via a network and, in particular, to methods and apparatus for downloading, authenticating and installing software components in a collaborative environment.

BACKGROUND OF THE INVENTION

[02] New collaboration models have been developed which operate in a "peer-to-peer" fashion without the intervention of a central authority. One of these latter models is built upon direct connections between users in a shared private "space". In accordance with this model, users can be invited into, enter and leave a shared space during an ongoing collaboration session between other users. Each user has an application program called an "activity", which is operable in his or her personal computer system, communication appliance or other network-capable device which generates a shared "space" in that user's computer. The activity responds to user interactions within the shared space by generating data change requests, called "deltas." The activity also has a data-change engine component that maintains a local data copy and performs the changes to the data requested by the deltas. The deltas are distributed from one user to another over a network, such as the Internet, by a dynamics manager component. When the deltas are received by another user activity in the shared space, the local data copy maintained by that activity is also updated.

[03] Subprograms within the activity program called "tools" perform various specific tasks. For example, an activity program might present various computer games. Within the overall game activity, tools might be available for a chess game and a tic-tac-toe game. Tools are defined by a template that is a document written in Extended Markup Language or XML. The XML template has various sections or tagged blocks that define various attributes of the tool that it defines.

[04] An application constructed in accordance with this model consists of a collection of software components and resources, all of which are targeted for execution on a particular device. As used herein the term "components" will be used to refer to both the software code and the resources that are required for an application. This type of collaboration system is described in detail in U.S. patent application serial no. 09/357,007 entitled METHOD AND APPARATUS FOR ACTIVITY-BASED COLLABORATION BY A COMPUTER SYSTEM EQUIPPED WITH A COMMUNICATIONS MANAGER, filed July 19, 1999 by Raymond E. Ozzie, Kenneth G. Moore, Robert H. Myhill and Brian M. Lambert; U.S. patent application serial no. 09/356,930 entitled METHOD AND APPARATUS FOR ACTIVITY-BASED COLLABORATION BY A COMPUTER SYSTEM EQUIPPED WITH A DYNAMICS MANAGER, filed July 19, 1999 by Raymond E. Ozzie and Jack E. Ozzie; U.S. patent application serial no. 09/356,148 entitled METHOD AND APPARATUS FOR PRIORITIZING DATA CHANGE REQUESTS AND MAINTAINING DATA CONSISTENCY IN A DISTRIBUTED COMPUTER SYSTEM EQUIPPED FOR ACTIVITY-BASED COLLABORATION, filed July 19, 1999 by Raymond E. Ozzie and Jack E. Ozzie and U.S. Patent application no. 09/588,195 entitled METHOD AND APPARATUS FOR EFFICIENT MANAGEMENT OF XML DOCUMENTS, filed June 6, 2000 by Raymond E. Ozzie, Kenneth G. Moore, Ransom L. Richardson and Edward J. Fischer.

[05] Computer software developers are often releasing new or upgraded versions of their software program components in order to correct problems or to add new features that were incomplete or unavailable when the software was originally released. However, in a collaborative system, such as that described above, the distribution of software is complicated by the dynamic nature of the collaborative activity. For example, a user who is invited to join a shared space may have a software component version that is incompatible with the version being used by other members of the shared space. Consequently, the user may not be able to interact with anyone in the shared space until his or her local software component (or components) is upgraded

to the current version and, since the user cannot interact with the shared space, the user may not be able to determine which component or components must be upgraded.

[06] In fact, the user may not know that a component upgrade is necessary until he or she is at the point of joining the shared space and, thus, the software upgrade may have to made unobtrusively while the user is waiting or using the collaboration system with other shared spaces.

[07] In addition, collaboration systems, such as that described above, are complicated and may comprise many software components. Many users will not be sufficiently knowledgeable to determine which software components need to be added, upgraded or replaced and to determine the locations at which the new components can be obtained. Some software components may incorporate other software that is located at yet other locations. The result is a complex and confusing download procedure. Other users will not be sufficiently knowledgeable to install and initialize those software components once they are obtained.

[08] In addition, since there is no central location with which all users must interact, there is also no central location from which software component upgrades can be released. Therefore, users may need to contact several locations, including download "farms", to obtain the necessary software components. Further, there is no central authority to enforce licensing restrictions or payment terms that may be imposed by some software developers on the transfer, or use, of software components that they developed. In addition, software components that incorporate other software components may require that licenses be obtained from several vendors, thereby making the licensing process complicated.

[09] Therefore, there is a need for a method and apparatus for downloading, authenticating and installing software program components over a network without user intervention.

[10] There is a further need for a method and apparatus for downloading, authenticating and installing software program components that can operate while the user is actively using a program that incorporates such components.

[11] There is also a need for a method and apparatus for downloading, authenticating and installing software program components from diverse locations.

[12] Finally, there is a need for a method and apparatus that can automatically enforce licensing restrictions without requiring that a user collaborate with the central location.

SUMMARY OF THE INVENTION

[13] In accordance with the principles of the invention, one embodiment of the invention uses a component manager that receives requests for component updates from a variety of sources, parses the requests and extracts URL information that identifies the location of a file containing the component resources. The component manager presents the URL to a download manager that asynchronously retrieves the component resources from the specified location and places the file in a staging area. Once the component resources have been downloaded, an install manager, also operating asynchronously from the component manager and the download manager, installs the component update.

[14] In one embodiment of the invention, the component manager maintains a list of all components installed on the system so that requested components already installed on the system are not downloaded again.

[15] In another embodiment of the invention, the component manager periodically polls component locations so that existing components can automatically be updated with new component versions.

[16] In yet another embodiment of the invention, the component manager validates downloaded component resources before the component resources are installed in order to prevent security breaches.

[17] In another embodiment, the component manager validates the root component in a component hierarchy and then extends a chain of trust from the root component to all other components in the hierarchy by processing each component.

[18] In still another embodiment of the invention, the install manager can install components using built-in installation programs or installation programs that are also downloaded from external sites.

BRIEF DESCRIPTION OF THE DRAWINGS

[19] The above and further advantages of the invention may be better understood by referring to the following description in conjunction with the accompanying drawings in which:

[20] Figure 1 is a block schematic diagram illustrating the main elements of the component manager, download manager and install manager.

[21] Figure 2 is a block schematic diagram of a client process and a master process illustrating the type and content of data persisted by the processes.

[22] Figure 3 is a flowchart illustrating the steps in an illustrative component update process.

[23] Figure 4 is a flowchart illustrating the steps in an illustrative process in which the component manager generates a component request to the download manager.

[24] Figure 5 is a block schematic diagram illustrating security elements used in validating and verifying a downloaded component.

[25] Figure 6 is a flowchart showing steps in an illustrative process for establishing an initial chain of trust for a downloaded root OSD file.

[26] Figure 7 is a flowchart showing steps in an illustrative process for extending the trust relationship from the root OSD file to any remaining components by resolving each component URL.

[27] Figures 8A, 8B and 8C, when placed together, form a flowchart showing steps in an illustrative process for signing each OSD file.

[28] Figures 9A and 9B, when placed together, form a flowchart showing steps in an illustrative process for installing system component updates in a collaborative system that is already running.

DETAILED DESCRIPTION

[29] The basic model used for component acquisitions and updates is a client server model in which the collaborative application acts as a client and retrieves software components from one or more servers. The client can connect to a component server over a network, such as a LAN or WAN, or the Internet. In the discussion below, the Internet will be assumed as the preferred network over which components are retrieved. Figure 1 shows the architecture of a component manager system constructed in accordance with the principles of the invention. As described in the aforementioned patent application, serial no. 09/357,007, a collaborator may have several shared spaces active in his or her device at any given time. Each shared space may be controlled by a separate client process of which processes 100 and 102 are shown in Figure 1. Other processes (not shown) may also be present. The device also has a single process, called a master process 100, that is responsible for the overall coordination of the system and the other process 102.

[30] In client process 102, two shared spaces 104 and 106 are illustrated in Figure 1, but more or less may actually be present. Similarly, although only a single shared space 108 is illustrated in client process 100, additional shared spaces may also be present.

[31] Component manager objects 110 and 112 are responsible for managing the retrieval, management and launch of the collection of software components required by the local collaborative installation. Each process with which a collaborator operates has its own component manager instance. For example, shared spaces 104 and 106 have component manager instance 112 and shared space 108 has component manager instance 110. Other instances may also be present, but are not shown for clarity.

[32] Each component manager object 110 and 112 receives "requests" for components and determines if, and when, the components need to be retrieved and installed on the local device. A request is a signal to the component manager that some

entity needs a specific component to be available on the local device. These requests can arrive from remote client installations, be locally created by the instantiation of a tool template, or be generated by the component manager itself as a result of polling for newer versions of already installed components.

[33] A special single component manager instance, called a system component manager 116, operates in the master process 100 and is forwarded all component update requests from all other component managers 110 and 112, via the component request queue 114. A local component manager, such as component manager 110 or 112, can determine if a component update request will result in a component download, but only the system component manager 116 actually performs downloads and installations. The system component manager 116 also handles tasks, such as polling for new versions of installed components on time-driven events associated with the installed components (for example, components can "expire" as discussed below). The system component manager 116 also coordinates the shutdown of certain processes that normally run continuously.

[34] The system component managers 116 works in conjunction with a download manager 130 and an install manager 132. The download manager 130 retrieves components designated by the system component manager 116 from a server over a network, such as the Internet, and the install manager 132 installs retrieved components in the local system. The system component manager 116 communicates with the download manager 130 via the download request queue 118, and communicates with the install request queue 120.

[35] The download manager part 130 of the master process 100 is responsible for servicing requests to retrieve software components from the Internet. As discussed below, each component is identified by a Uniform Resource Locator (URL) that is passed to the download manager 130 by the system component manager 116 to begin the download process. The download manager 130 works with the network transport layer of a network protocol stack in order to retrieve components without interfering with the normal transfer of information between collaborators, via deltas. The system

component manager 116 and the download manager 130 communicate by exchanging information via the download request queue 118 which can be an element queue as described in aforementioned application, serial no. 09/588,195.

[36] The install manager 132 is the part of the master process 100 responsible for servicing requests to install components once the components have been downloaded. Because installations must be serialized and since multiple component managers can be working independently, the install manager 132 exists in the master process 100 in order to coordinate installation requests. As with the download manager 130, the system component manager 116 and the install manager 132 communicate through a queue 120 that can be an element queue. Component managers 110 and 112 also implement factories which instantiate components in order to satisfy activation requests for components employed by a given shared space.

[37] A system component installer 136 handles installation components which cannot be installed while any other collaborative process is active. These components are often system components which comprise the collaborative platform but they can also be tools which are currently in use. The former components could include, for example, a storage manager, a communications manager, and the component manager, download manager and install manager themselves. Also included are any components that are essential to the collaborative system and for which on-demand loading would be difficult. These former components might include a dynamics manager, member manager or activity manager. The system component installer 136 does not rely on any other software so that it can upgrade system components when the rest of the collaborative system is unloaded. During operation, the system component installer 136 is launched by the install manager 132. When it has finished running, the system component installer 136 launches the normally running portion of the collaborative system.

[38] The system component manager 116 and the system component installer 136 maintain persistent data that relates to the components to be installed and to the components already installed. Figure 2 illustrates this persistent data. In particular,

Figure 2 illustrates a master process 200 and a system component installer 202. The master process 200 includes a shared space 204, a component manager 206, a system component manager 208 and an install manager 210.

[39] Both the component manager 206 and the system component manager 208 consult the manifest, which is a persistent XML document maintained in storage 212. The component manager 206 inspects the manifest, as indicated by arrow 230, when the component manager 206 receives a component request, for example from shared space 204, to determine if the component is already installed on the current device. Component manager 206 also consults the manifest if it is called to instantiate a component. For example, if shared space 204 required components A and B, the component manager 206 could determine from the manifest that both component A 214 and component B 216 were already installed. Furthermore component manager 206 could instantiate these components if requested to do so.

[40] The system component manager 208 also consults the manifest in storage 212, as indicated by arrow 232, to determine when to poll for new versions of components or when to deem components to be expired. When a new component is added to the device, the system component manager 208 is responsible for updating the manifest.

[41] When a component manager 206 receives a request for a component which is not installed, it creates a component update list (CUL) in an "in progress" component update requests document in storage 218 as schematically indicated by arrow 234. The system component manager 208 processes component update requests in the in progress document 218, as schematically indicated by arrow 236, and periodically updates the document with the progress of each component update request. In Figure 2, there are two requests in progress, Component C 220 which has just been added and Component D 222 for which the system component manager 208 has used the download manager (not shown in Figure 2) to retrieve the component OSD file.

[42] Components often express dependencies on other components so that a CUL usually begins with a single component URL and then grows into a tree of component descriptions. An error during the downloading, verification or installation of any of these components requires that the downloading, verification and installation of the entire tree of components be aborted without causing major problems.

[43] A separate file called a system component install queue is used to communicate between the system component installer 202 as indicated by arrow 226 and the install manager 210 as indicated by arrow 228. The system component install queue file is typically stored in a storage area 224 that is associated with the master process 200. The file contains a list of components that the system component installer 202 must install and the results of the installation process for each component once the system component installer 202 has attempted the installations. This list contains all the components which have been, or will be, installed as part of processing the current CUL. In the case of failure during the installation of some component, this list is used to roll back any installations associated with the CUL. The list can be XML based as illustrated in Figure 2.

[44] The process that occurs once a request for a new or updated component has been received by the system component manager is called a component update. An overview of the process is shown in Figure 3. The process commences in step 300 and proceeds to step 302. There are six phases to the component update process: component request 302, component discovery 303, component download 304, component verification 306, component installation 308 and component publish 309. The process then finishes in step 310.

[45] The first four phases, 302, 303, 304 of a component update occur in an asynchronous fashion and sometimes only when the system has idle periods. They generally do not involve user intervention. The verification step 306 and the installation step 308 can require some user input and parts of the collaborative system may depend on its completion. For example, the user may be asked to trust the component developer or to answer some prompts generated by an install program.

[46] The first phase of a component update involves the system component manager 116 being alerted that a component needs an update. The system component manager 116 both polls for, and is asynchronously notified of, the need to update components. In the case of asynchronous notification, the component manager either receives a "component update delta" or an XML-RPC message that causes the component manager to initiate a download. In the case of polling, the component manager uses attributes of a locally-installed component to determine how often to check for a newer version. Thus, polling only applies to already installed components while notifications can apply to new or already installed components.

[47] More particularly, there are at least four ways to trigger the update process. These include a user generated update request in which a user (or process) generates a request by performing one of several actions. For example, a user could start the collaborative system for the first time, the user could add a new or updated tool to a shared space or the user could invite a new user to join a shared space. In the case of a user generated update request, a local component update delta is generated and then dispatched to other shared space members. When a new user is invited to join a shared space, the new member receives a copy of the shared space which incorporates a list of required components. These are both examples of a synchronous notification.

[48] The second way to trigger an update process is by polling. Frequently, the system component manager 116 will inspect its local manifest of installed components to see if any component has reached an update "time" that is defined by the component provider. In addition, the system component manager 116 can cause a download of a component file from the location from which the component was originally retrieved and check to determine whether the component file references a newer component. If so, the system component manager 116 will cause the newer component to be downloaded as well.

[49] The third way to initiate an update process is for a user to "inject" a component by executing a specific program file type. This file type can be interpreted

by the collaborative system and, if the contents of the file specify a component request, the contents will be routed to the system component manager 116 which will begin a component update, if necessary.

[50] The fourth way to trigger the update process is by an external agent. In rare cases, there may be a need to "alert" the system component manager 116 that it needs to get a component immediately (for example, a component was found to have a security problem, but the update age had not been reached). In this case an "alert bulletin" could be registered with a server that polls frequently for various types of bulletins. This server could then download the alert and send it to the system component manager 116.

[51] Component resources are any files that are downloaded, installed, and managed by the component manager and are described in a language called an "open software description" or OSD. This language is contained in a proposal before the World Wide Web Consortium to describe the components of a software package in XML using predefined tags. Component resources are defined within an OSD file by using a <SOFTPKG> tag. A hierarchy of component resources can be defined using the <DEPENDENCY> tag in which another <SOFTPKG> tag is nested. A <SOFTPKG> block in an OSD file instructs the component manager to manage these OSD files. The details of the OSD language can be found on the World Wide Web Consortium web site http://www.w3.org/TR/NOTE-OSD.html and are incorporated by reference herein.

[52] In the inventive system, the basic OSD language is used with extensions to describe software components and an OSD description can refer to other OSD descriptions. Although component resources are defined as nothing more than files, they often are, or contain, components. As discussed below, version compatibility and COM requirements apply to component resources that are, or contain, components. These requirements do not apply for other types of component resources, such as WAVs or GIFs.

[53] Generally, components defined in a template do not correlate one-to-one with component resources as defined in an OSD file. However, there are two common

ways they are related. In the first way, many components, such as COM objects, are related to one component resource, such as a DLL file. In particular, many COM objects that are individually listed in a template can be put into a single DLL file. For example, a DocumentShareTool.DLL could contain DocumentShareView, DocumentShareEngine, DocumentShareCode, DocShareReplaceFileCode, DocSharePropertiesCode, and DocShareCreateLinkWizardCode objects. In the DocumentShareTool template, all of these objects are defined in their own <Component> tagged block. Each of these tagged blocks then contains a similar component resource URL because all of these components need to point to the same component resource - the DocumentShareTool.DLL file. However, they often include a unique factory which maps to an instantiatable object within the DLL.

[54] Another example is a Games.DLL component resource that contains Chess and Tic-Tac-Toe objects. In this example, there are more components in the component resource than are required by a single tool. Adding the Chess tool to a shared space has the side effect of installing the component resources needed for the Tic-Tac-Toe tool. This is because the Games.DLL is a dependency for both the Chess Tool and the Tic-Tac-Toe tool.

[55] The other common scenario is a single <Component> tagged block in a template which points to an OSD file that defines multiple component resources. For example, the component may be included in a DLL file, which relies on other support files like a utility DLL file, some GIF files, and a WAV file. These ancillary component resources in the OSD file may be defined individually so that they can be individually managed by a component manager. For example, assume that one of the dependencies of a COM object specified in a tool is a DLL file which contains functions used by other COM objects that are not part of the tool. By making the component manager aware of the DLL file, the installation and download of the other COM objects could be expedited.

[56] A component update request that a component manager receives asks for a component resource by providing a URL that identifies the resource to the component

manager. Component resource URLs always point to the Internet, via an HTTP or an FTP scheme. The format of a component resource URL has two parts separated by a "?" character. The following is a simple example:

ftp://components.groove.net/Games.osd?Package=net.groove.Chess&Version=2

[57] The first URL part that consists of text to the left of the "?" character specifies an OSD file. The OSD file can be self-contained or it can reference, recursively, other OSD files. The second part of the URL, the text to the right of the "?", provides a query that the component manager uses to evaluate the OSD file(s). Within the query portion of a component resource URL there is a token "Package" followed by an operator "=", in turn, followed by a string value which is the name of a "package." A package logically defines the set of functionality that makes a component resource unique. Consequently, although a package can have multiple implementations, the functionality of the package is designed to be roughly constant. Package names must be unique for all component resources and are used by a component manager to determine whether a component with the required functionality is installed in the shared space. To accomplish this, the package name can be hierarchical or can be a guaranteed universal identifier. In the above example, the package "net.groove.Chess" represents a chess tool.

[58] The OSD description for a component resource contains information defining the component resource version. A component resource version is loosely defined in the OSD language as a string containing four, comma-separated, version values: Major, Minor, Custom, and Sequence. A major version value defines specific features that differ among the other major versions of a package. These features must be additive such that a later major version will support all the features in an earlier major feature (that is, major versions are backward compatible). A later major version is required to understand any persistent data written by an earlier version. Two major

versions are not expected to be able to communicate with each other either (that is, exchange deltas). Different major versions must have different COM class identifiers (CLSIDs) and, as such, can coexist on the same device. This allows a single shared space, for example, to be using two different major versions of a tool (separate tabs) which are logically the same tool.

[59] Because major versions define feature sets and compatibility for a package, a client requesting a component resource will often specify a major version as part of the query in the component resource URL. In the example, URL set forth above, the query requests a component resource with a major version equal to version 2.

[60] Minor, Custom and Sequence values all have the same basic semantics - new minor, custom, and sequence number versions within the same major version are required to be forward, and backward, compatible. They need to work with persistent data created by, and to communicate with, any other versions of the same major version. Only one instance of a major version can be installed at a time on a given device so a change in any of these three version values cannot change the CLSID.

[61] For simplicity, the discussion below will use the term "minor version" to mean any of these three version type values. The purpose of these version types is to fix bugs and not to introduce features, and there are few reasons why a tool/template developer would specify a minor version in a component resource URL. However, if such a minor version is specified, the minor version is always interpreted as the specified version or a later version.

[62] The request step is illustrated in more detail in Figure 4. The process begins in step 400 and proceeds to step 402 where the URL in the request is parsed to extract the package name. In step 404, the package name is compared to the list of package names on the manifest. Based on this comparison, in step 406, a decision is made whether the package must be downloaded. In many instances, a component update request does not result in a component download because a suitable version of the component is already installed on the local device. For example, assume the

component resource specified by the example URL set forth above was already installed, and a subsequent request was received with the following URL:

http://www.tucows.com/Components/Groove/Games.osd?Package=net.groove.Chess

[63] Even through this subsequent URL points to a different OSD file, the component manager would not have to process the OSD file because it already has the net.groove.Chess package. The capability of recognizing a package by its name allows tool/template developers to mirror OSD and component resources. Note that, in order to make the query work, the name used for the package must match the NAME attribute of the <SOFTPKG> tag of the OSD file referenced by the URL.

[64] Not every component resource that is defined in OSD needs to be in a separate downloadable file. For example, say that a component resource, A.DLL, depends on ancillary component resources bundled in a single CAB file. In this case, the <CODEBASE> tag, which specifies the downloadable file, would only be present in one of the component resources. This allows a component manager to manage the individual component resources that are in a CAB file.

[65] If, as decided in step 406, a suitable component has already been installed, then, in step 408, the URL provided with the request is mapped to the existing component. Alternatively, if, in step 406 it is determined that a suitable component is not installed, then the URL is passed to the system component manager to obtain the component.

[66] In the second phase of the component update process - component discovery 303 - the system component manager downloads and parses the OSD file specified by the component resource URL. The goal of the component discovery is to find the <SOFTPKG> element that describes the component named in the query. There is no guarantee that the correct <SOFTPKG> element will reside in the OSD file

specified by the URL. It is possible to "redirect" the system component manager to a new OSD file with the following syntax:

<SOFTPKG NAME="net.groove" VERSION = "0,0,0,0"
Lg:Redirect HRef=http://componenets.groove.net/Groove.osd/>
</SOFTPKG>

This feature is extremely useful because it allows component developers to re-organize their OSD without invalidating published component resource URLs.

[67] Once the required <SOFTPKG> element is found, the system component manager determines if the component has any dependencies which are not installed on this device (that is, not listed in the manifest). If so, the same process of discovery is reclusively applied to this dependency. In this fashion, the system component manager constructs a tree of <SOFTPKG> elements that need to be downloaded and installed.

[68] During the component discovery phase, the system component manager also may apply "device policies". A given set of collaborative system devices can be specified to exist within a management domain that is managed by an administrator. The administrator of this management domain can specify "device policies" that allow or prevent the download and installation of components on the managed devices. These device policies are installed on each client in the management domain using an XML-RPC method.

[69] In the component discovery stage, the system component manager inspects the current device policies to determine if they are applicable to the present component update. Explicit permission to install a given component can be specified in a device policy and will cause the system component manager to omit asking for the user's permission. Explicit prohibition to install a component can also be specified in a device policy and will cause the system component manager to abort the component update.

[70] The third phase of a component update 304 (Figure 3) involves the download manager, having been called by a component manager, retrieving the component identified by the OSD file and placing the retrieved component in a staging area. The download operation is asynchronous with the operation of the system component manager and the remainder of the system. First, the download manager determines whether the component has already been downloaded. There are many situations where the system component manager issues a download request and then terminates. Since the download is asynchronous, downloading will continue to completion even in the absence of the system component manager. However, the download manager cannot notify the system component manager that the download is complete because the system component manager does not exist. When that system component manager restarts, it will re-issue the download request at which point the download manager can inform the system component manager that the component has already been downloaded.

[71] The download manager operates with a "transport layer" layer to perform the actual download operation. The transport layer provides an interface to download a file based on a URL. In general, the download manager can initiate more download requests than the transport layer will allow, in order to prevent interference with the collaborative system. Therefore, the download manager can expect the transport layer to slow those requests as needed.

[72] In a preferred embodiment, the download operation will be capable of stopping when a file is partially downloaded and then restarting without starting the download at the beginning of the file. Thus, the transport interface must take a file offset from which to restart. On initial download the offset will be zero. The transport layer writes the file to a stream object as the file is downloaded. The transport interface may also take flags to specify certain download requirements. For example, a priority flag may be specified to help the transport layer determine how aggressive the operation should be. In order to retrieve the file, the transport layer will try different protocols in some order based on efficiency.

[73] After calling the transport interface, the download manager thread reads from the stream pending, if necessary and writes the stream data to a file in the download manager staging area. The name of the file is determined by the components OSD name and the staging area is partitioned by the URL path. The download manager also notifies the system component manager of its progress, via an element queue, if requested. Finally, when the download manager thread receives an end-of-stream return code, it closes the staging file and notifies the system component manager that the file transfer, but not the file verification, is complete. In a preferred embodiment, components can be downloaded from "component farms", which are commodity servers that host components available for download by the component manager.

[74] The component discovery phase employs the download manager to download OSD files in the same fashion that the download phase employs it to download components. However, the user rarely perceives the download of OSD files, as they may perceive the download of components, because the OSD files are generally much smaller.

[75] The fourth phase of a component update 306 involves the system component manager authenticating the integrity of the file just downloaded. Component authentication is performed by the system component manager after a component has been downloaded, but before the system component manager installs the component. Although each component is individually authenticated, there is a chain of trust that extends from a root component through each child component in the component hierarchy.

[76] Figure 5 illustrates a hypothetical component hierarchy. In this diagram, there are four OSD files 500, 502, 504 and 506. OSD file 500 is associated with component 508 that is to be installed. Similarly, OSD file 502 is associated with component 510 and OSD files 504 and 506 are associated with components 512, 514 and 516, 518, respectively. Each OSD file contains the component security data elements that are needed for the set of components in the OSD file. For simplicity, the

OSD data elements (for example, SOFTPKG, DEPENDENCY) that would be required by this hierarchy have been omitted.

[77] There are five types of security data elements possible in any given OSD file, for example, OSD file 500. These include a single signature (certificate and encrypted digest) 520 used to authenticate the contents of the OSD file 500. A single certificate 522 used to validate the signer is included. Also present is a fingerprint table 524 that contains one fingerprint for each unique signer of OSD files referenced by the current OSD file 500. A fingerprint is the digest of a certificate. In OSD file 500, each entry in the fingerprint table 524 contains a simple name for the fingerprint , for example, name 526, as well as the corresponding fingerprint itself 528. Similarly, name 530 is associated with key 532.

[78] Secure component URLs may also be included. A secure component URL is the pairing of a normal component URL (such as URLs 536 and 544) and a simple name (534, 542) that maps to a fingerprint table entry. In other words, a secure component URL is a component URL and a fingerprint. The fingerprint table 524 is merely a convenience to save space in the OSD file. Secure codebase URLs can also be included. A secure codebase URL is the pairing of a normal codebase URL 540 and a digest 538 of the component 508 that is referenced by the codebase URL 540.

[79] These security data elements are used to guarantee the authenticity of the components in two distinct ways. First, the signature 520 on the OSD file 500 and the digest 538 of the codebase 508 are used to guarantee that the contents of the OSD file 500 and the codebase 508 exactly match their contents at the time they were signed. The reason this is important is because both OSD file 500 and codebase 508 may be downloaded off a public network, such as the Internet, and are susceptible to alteration (both at the site where they are stored or via spoofing). The component manager can verify the signature and the digest in a conventional manner in order to guarantee that these entities have not been altered.

[80] Secondly, the public keys 534, 542 contained in the secure component URLs is used to guarantee that the organization, or individual, ("B") that signed an OSD

file, such as OSD file 506 is the same organization, or individual, that the signer of a referring OSD file 500 thinks signed the file 506. This is a very different type of authenticity than in the first case. The public key 532 cannot provide any information about the contents of the OSD file 506 to which it refers so the contents may be very different between the time when the referring OSD file 500 was signed (by "A") and the time that a component manager attempts to authenticate the OSD file 506. Therefore, it is the signer ("B") who is trusted rather than the contents of the secondary OSD file 506 and the authentication relies on a chain of trust.

[81] In the specific hierarchy shown in Figure 5 the root OSD file 500 is signed 520 by organization "A" and contains a single component 508 which depends on two other components 510 and 516. The first 510 of those components is in an OSD file 502 signed by "A" and the second component 516 is in an OSD file 506 signed by "B". In turn, OSD file 502 contains a single component 510 which depends on two other components 512 and 514 which are contained in an OSD file 504 signed by "C". OSD file 504 contains two components 512, 514 with no dependencies on other components. Thus, the public key table 550 is empty. Similarly, OSD file 506 is signed by "B" and again contains two components 516 and 518 with no dependencies.

[82] In accordance with a preferred embodiment, component authentication is accomplished by establishing the chain of trust from the root component 508 downward through the hierarchy. To accomplish this, the component manager treats the root component 508 in special manner that allows it to assert the initial trust relationship. Once that trust is established the component manager can use the security data elements described above to extend the trust down the component hierarchy.

[83] Every component update requests begins when some object calls the component with a component URL. This first component URL 560 is identical in format to the component URLs (for example, URLs 536 and 544) that point from one OSD file to another. However, component URL 560 is special because it specifies the root of the component hierarchy from which the chain of trust will be established. It is also special in that it is not a secure component URL. The steps for establishing this initial trust are

illustrated in Figure 6. The process begins in step 600 and proceeds to step 602 in which a component manager, having determined that the component specified by the initial component URL needs to be installed, uses the download manager to download the OSD file 500 specified in that URL 560.

[84] Next, in step 604, the component manager extracts the certificate 522 from the downloaded OSD file 500 and either validates the certificate itself or provides the certificate to another process, such as a security manager, for validation. Validation entails checking the certificate information for correctness and validity (for example, expiration dates).

[85] In step 606, the process determines if the extracted and validated certificate is in the user's trusted key ring. If it is, then the signer can be trusted and the process proceeds to step 612. If it is not, then the certificate information is displayed to the user in step 608 and the user is asked if they trust the signer in step 610. If the user does not trust the certificate then the process finishes in step 614. Alternatively, if the user does trust the certificate, the process proceeds to step 612.

[86] Once the certificate is validated, in step 612, the component manager verifies the signature 520. To do this, the component manager first removes the signature 520 from the OSD file 500 because the signed data set does not include the signature 520. The component manager then computes the digest of the modified OSD file 500 and verifies the signature 520 using this digest and the public key in the certificate 522.

[87] Assuming all goes well the process finishes in step 614 and it has been established that the root OSD file 500 is byte-for-byte the same as it was when it was signed and that the user trusts the person that signed it. The OSD file 500 can now be parsed.

[88] The previous steps described the special handling the component manager applies to the root OSD file 500 to assert the initial trust relationship. Figure 7 illustrates the steps in a process that the component manager uses to extend that trust relationship to the remaining components by resolving each component URL. These

steps are applied recursively from the root component through all child components. This process starts in step 700 and proceeds to step 702 where the next secure URL in the chain is retrieved. In step 704, a decision is made whether the secure URL refers to a component codebase or to another OSD file.

[89] If the URL refers to a secure codebase, then, in step 706, the codebase is downloaded. In step 710, the digest of the codebase is computed and in step 714, the computed digest is compared to the stored digest.

[90] Alternatively, if the URL refers to another OSD file, then, in step 708 the OSD file is downloaded. In step 712, the digest is computed (minus the signature). Next, in step 716, the signature is verified using the public key from the referring OSD file and the just computed digest. Using the public key from the referring OSD file is important because this key guarantees that the signer of the secondary OSD file is who the signer of the referring OSD file thinks it should be. Simply using the public key from the certificate of the secondary file would not provide the same referential integrity.

[91] In the case of either type of URL, the process proceeds to step 718 where a determination is made whether additional URLs remain to be processed. If so, the process proceeds back to step 702 whether the next URL is retrieved. If not, the process finishes in step 720.

[92] Most of the time when a component manager is requested to resolve a component URL, the component in question has already been installed so that the component manager does not need to download the component. However, with secure component URLs, the component manager can also make sure that any previously-installed component was signed by the same organization by which the current secure component URL claims it has been signed. To support this feature, a component manager can store in every component entry in the manifest, the fingerprint of the signer of that component. Then, when a component manager checks to see if a component is already installed, it can also compare the manifest fingerprint against the secure component URL fingerprint.

[93] When fingerprints are included in the manifest, it could be the target of an attack to compromise component security. To prevent this the manifest can be message authentication coded with a secret key which is unique (generated) for each client. Whenever the manifest is altered, it must be coded. However right before being altered, the previous message authentication code is re-verified to ensure that the only change going into the manifest is the one being added in this operation. In general, semaphores or some other mechanism can be used to guarantee that the re-verify, apply change, and recoding operation is atomic.

[94] When the collaboration system is first installed, a pre-seeded manifest is also installed, which reflects the components that are part of the installation kit. Since the secret key with which the manifest is signed is generated on the client, the manifest in the installation kit cannot be coded. Instead, it can be signed. Accordingly, when a component manager first accesses the manifest it will discover that the manifest is signed (rather than message authentication coded) and will generate a unique secret key, and then message authentication code the manifest.

[95] If the authentication of an OSD file or codebase fails during any point of aforementioned process, the component update is halted and the update subscriber is notified.

[96] Component signing is the process by which the various component security data elements described above are applied to OSD files. In one embodiment, a component signing utility works on a single OSD file at a time. Although the utility may be called with a set of OSD files, the utility will behave as if it were called separately for each OSD file. Because the OSD files are processed individually, the order of the signing is insignificant. The following is an illustrative signing algorithm. Other algorithms would be apparent to those skilled in the art.

[97] Referring to Figure 5, in order to simplify the utility implementation, there is a clear division between security data elements constructed by the OSD file author and elements generated by the signing utility. Specifically, with respect to OSD file 500, the URLs 536, 544, the public key names 534 and 542 and the public key table 524 and its

contents are data elements that are provided by the OSD author. The digests 538, certificate 522, and signature 520 are added by the signing utility.

[98] With this division, the signing utility will take as arguments one or more OSD files; an output directory name, the signer's certificate file name and the signer's private key file name. Figures 8A-8C, when placed together, form a flowchart that shows the steps in the illustrative process that the component signer can use for each OSD file provided. This process starts in step 800 and proceeds to step 802 in which the OSD file is imported into an XML document. In step 804, a conventional locator subroutine is used to find the next codebase URL in the XML document.

[99] In step 806, the codebase specified by the located URL is downloaded. Next, in step 808, the digest of the codebase file is computed. Then, in step 810, the computed digest is inserted into the OSD file, making the codebase URL into a secure codebase URL. A check is made in step 812 to determine whether any more codebase URLs exist in the document. If there are more codebase URLs, then the process proceeds back to step 804 to process the codebase URLs. If there are no more codebase URLs as determined in step 812, the process proceeds, via off-page connectors 814 and 816, to step 818.

[100] In step 818, the locator is used to find the next secure component URL in the XML document. In step 820, the public key table is checked to make sure that it contains the public key name of the secure component URL. If, as decided in step 822, the public key table does not contain the fingerprint name, the fingerprint name is added in step 824 and the process proceeds to step 826. Otherwise, the process proceeds directly to step 826.

[101] In step 826, a determination is mode whether additional secure component URLs exist in the XML document. If so, the process proceeds back to step 818 so that each secure component URL is processed. Otherwise the process proceeds, via off-page connectors 828 and 830, to step 832.

[102] Next, in step 832, any unused fingerprints are deleted from the public key table. Similarly, in step 834, if a certificate exists in the XML document, it is also

deleted. Next, in step 836, the new certificate, provided as an argument, is optionally validated and inserted into the XML document. In step 838, any previous signatures are deleted. In step 840, a new signature is computed and inserted into the XML document. Finally, in step 842, the XML document is serialized to the same file name in the output directory and the process finishes in step 844.

[103] The fourth phase (310, Figure 3) of component update involves a component manager calling an install manager (132, Figure 1) to execute an installation program specified in the OSD file of the downloaded component. For example, within a <SOFTPKG> block of the OSD file, a component developer can specify how to install the component resource using an <Install> tag. The installation phase is usually triggered by a user action such as the opening a shared space with an old component.

[104] The installation program may be one of a set of installation programs that are provided with the install manager. When a component manager, such as component manager 110, reaches the install phase for a specific component resource, it calls the install 132 manager and passes in the <Install> tag from the component resource's <SOFTPKG> block. The install manager then interprets the XML statements in the <Install> tag block and, in the case of built-in installers, runs the installer code in-line.

[105] There can be several types of built-in installers. An illustrative example is an "InProc Server" installer that copies the downloaded file from the staging area either to a predefined executable directory or to a specified target directory and then loads the DLL, and calls its DllRegisterServer entry point. Another built-in installer called a "Copy" installer could simply copy the file from the staging area to the predefined executable directory unless another target directory is specified. The following are XML statements for the previous examples.

```
<Install Type="InProc Server"/>
<Install Type="Copy" TargetDir="c:\Program Files\My App"/>
```

[106] Alternatively, the installation program may be a third party application which itself must be downloaded. If a component developer requires a custom installer, the installer would be downloaded by making it a dependency of the to-be-installed component resource and then specifying the installer in the <Install> tag for the component resource. In general, an external installer is a stand-alone executable that the installation manager launches. When a component manager launches an external installer, it can provide a number of different parameters on the installer's command line. The model is basic keyword substitution. Whatever part of the command line is not keywords is passed as is to the installer intact. Some possible keywords are:

$STAGINGFILE$	the staging file for the current <SOFTPKG>
$STAGINGDIR$	the root of the staging file directory
$TEMPDIR$	the system temp directory
$GROOVEBIN$	the predefined executable directory
$GROOVEDATA$	a data directory

[107] Some mechanism must also be used to determine whether the installation program has successfully completed. Illustratively, this can be performed by launching each external installer on a newly spawned thread. That thread then performs a WaitOnSingleObject for the installer to complete. When the installer exits, the component manager captures the process exit code. A non-zero code is considered to be an error and will prevent any dependent component resources from being processed.

[108] Components may be installed on a device without the knowledge of the system component manager. For example, a Shockwave player can be installed directly by a user or by the system component manger in response to the use of Shockwave in a shared space (SHOCKWAVE is a trademark of Macromedia, Inc. 600 Townsend Street, San Francisco, CA 94103.) Continuing this example, assume that

Shockwave has been installed directly by the user and that the system component manager has just received a request for that same component. Since Shockwave is not listed in the Manifest, the system component manager would normally download and install Shockwave again, needlessly consuming the user's network connection. However, the author of the OSD file can specify a <g:InstallTest> element which will cause the system component manager to first test for the existence of Shockwave before downloading it. If it does exist, the system component manager skips to the component publish phase. An install test is also useful for external installers which might have failed or been canceled without the knowledge of the system component manager. Some example types of InstallTests include CoCreate tests, file exists tests, file date time tests and registry tests.

[109] The installation of collaboration system components (for example, the storage manager) is accomplished with a system component installer (136, Figure 1.) The system component installer 136 is unique because, in order to install system components or other in-use components, the collaborative system itself cannot be running. Thus, a thread cannot be used to track the completion of the installation process. Figures 9A and 9B, when placed together, form a flowchart that shows the steps in an illustrative process for installing in-use component updates in a collaborative system that is already running. The process begins in step 900 and proceeds to step 902 where the system component manager 216 (Figure 2) running in the master process 214 discovers the need for new in-use components.

[110] Next, as shown in step 906, the system component manager 216 downloads the new components, copies them to a pre-defined directory (such as the system component list 228 as indicated by arrow 224) and updates the persisted system CUL 234. In step 907, the Install Manager attempts to install a component and discovers it to be in-use. It next enqueues the installation to the system installer. In fact, all subsequent installations that are part of the current CUL must be enqueued to the system installer once the first in-use component is discovered. This requirement is born of the necessity to guarantee the order in which the components are to be

installed. Once all of the installation requests for the current CUL have been enqueued to the system installer, the install manager 132 prompts the user to restart the collaborative system. Until that happens, all system component manager activity is halted.

[111] Then, in step 920, the system component installer 222 installs all the system components. In step 922, the system component installer 222 updates the install queue document 224 indicating that the installation process has been completed, including any error information. The system component installer 222 then starts the collaborative system and terminates in step 924. In step 926, the master process 214 (and the system component manager 216 in it) restarts and the system component manager 216 finds in the persistent storage area 220, a half-completed system CUL 234 for the system components. When it starts, the system component manager 216 restarts where it left off processing the system CUL 234. The system component manager 216 discovers, in step 928, that the install queue has been updated and completes the component update by writing the new components to the manifest 236 and deleting the system CUL 234. The process then finishes in step 930.

[112] The instantiation of an installed component can be requested by some higher component. The component manager supports component activation by invoking the appropriate "factory" for the requested component. A factory is simply an object that can instantiate certain other kinds of objects. Factories have a well-defined interface that the component manager understands. The interface between the factory and the requested component is usually designed by the factory/component creator.

[113] The collaboration system can include several built-in activation factories but third parties can also download their own factories provided the new factory is launched by one of the built-in factories. Some components, for example GIFs, will not be activated by the collaboration system but merely used directly by some other component. Any given version of a component will most likely be activated many times but only downloaded and installed once.

[114] A software implementation of the above-described embodiment may comprise a series of computer instructions either fixed on a tangible medium, such as a computer readable media, for example, a diskette, a CD-ROM, a ROM memory, or a fixed disk, or transmittable to a computer system, via a modem or other interface device over a medium. The medium either can be a tangible medium, including but not limited to optical or analog communications lines, or may be implemented with wireless techniques, including but not limited to microwave, infrared or other transmission techniques. It may also be the Internet. The series of computer instructions embodies all or part of the functionality previously described herein with respect to the invention. Those skilled in the art will appreciate that such computer instructions can be written in a number of programming languages for use with many computer architectures or operating systems. Further, such instructions may be stored using any memory technology, present or future, including, but not limited to, semiconductor, magnetic, optical or other memory devices, or transmitted using any communications technology, present or future, including but not limited to optical, infrared, microwave, or other transmission technologies. It is contemplated that such a computer program product may be distributed as a removable media with accompanying printed or electronic documentation, e.g., shrink wrapped software, pre-loaded with a computer system, e.g., on system ROM or fixed disk, or distributed from a server or electronic bulletin board over a network, e.g., the Internet or World Wide Web.

[115] Although an exemplary embodiment of the invention has been disclosed, it will be apparent to those skilled in the art that various changes and modifications can be made which will achieve some of the advantages of the invention without departing from the spirit and scope of the invention. For example, it will be obvious to those reasonably skilled in the art that, in other implementations, different arrangements can be used for the scope and arrangement of the federated beans. Other aspects, such as the specific process flow, as well as other modifications to the inventive concept are intended to be covered by the appended claims.

[116] What is claimed is: